US005848274A

United States Patent [19]

Hamby et al.

[11] Patent Number: 5,848,274

[45] Date of Patent: Dec. 8, 1998

[54] **INCREMENTAL BYTE CODE COMPILATION SYSTEM**

[75] Inventors: **John Hamby**, Issaquah; **Niklas Gustafsson**, Bellevue; **Patrick Lau**, Renton, all of Wash.

[73] Assignee: **Supercede, Inc.**, Bellevue, Wash.

[21] Appl. No.: **645,955**

[22] Filed: **May 10, 1996**

Related U.S. Application Data

[63] Continuation-in-part of Ser. No. 608,820, Feb. 29, 1996, Pat. No. 5,764,989.

[51] **Int. Cl.**[6] **G06F 9/45**

[52] **U.S. Cl.** **395/705**; 395/701; 395/704; 395/706; 395/685

[58] **Field of Search** 395/701, 702, 395/704, 705, 707, 710, 685, 706

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,802,165	1/1989	Ream	371/19
4,809,170	2/1989	Leblang et al.	364/200
4,953,084	8/1990	Meloy et al.	364/200
5,170,465	12/1992	McKeeman et al.	395/700
5,175,856	12/1992	Dyke et al.	395/700
5,182,806	1/1993	McKeeman et al.	395/700
5,193,191	3/1993	McKeeman et al.	395/700
5,201,050	4/1993	McKeeman et al.	395/700
5,204,960	4/1993	Smith et al.	395/700
5,265,254	11/1993	Blasciak et al.	395/700
5,291,583	3/1994	Bapat et al.	395/500
5,307,499	4/1994	Yin et al.	395/700
5,325,531	6/1994	McKeeman et al.	395/700
5,327,562	7/1994	Adcock et al.	395/700
5,339,431	8/1994	Rupp et al.	395/700
5,339,433	8/1994	Frid-Nielsen	395/700
5,355,494	10/1994	Sistare et al.	395/700
5,357,628	10/1994	Yuen	395/575
5,371,747	12/1994	Brooks et al.	371/19
5,375,239	12/1994	Mortson	395/700
5,426,648	6/1995	Simamura	371/19
5,432,795	7/1995	Robinson	371/19
5,459,868	10/1995	Fong	395/700
5,590,331	12/1996	Lewis et al.	395/708

FOREIGN PATENT DOCUMENTS

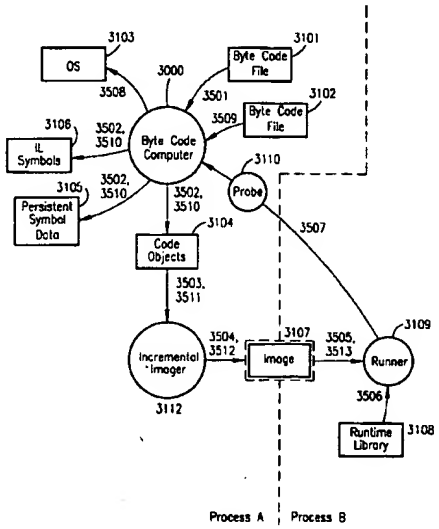
90307228	2/1990	European Pat. Off. .
91115971	9/1991	European Pat. Off. .
2701580	10/1993	France .
H6-83597	3/1994	Japan .
H6-161726	6/1994	Japan .
H6-266563	9/1994	Japan .
H6-274349	9/1994	Japan .
PCT/US91/04064	6/1991	WIPO .
PCT/US93/05368	6/1993	WIPO .
PCT/US94/00041	6/1993	WIPO .
PCT/US94/00342	6/1993	WIPO .

Primary Examiner—Emanuel Todd Voeltz
Assistant Examiner—Kakali Chaki
Attorney, Agent, or Firm—LaRiviere, Grubman & Payne

[57] **ABSTRACT**

An incremental byte code compiler which provides a high-performance execution environment for dynamically linked languages and for distributed target-independent applications. The execution environment provided by the present invention includes an incremental byte code compiler for generating IL symbols and code objects from a byte code source file, a persistent symbol table for storing the IL symbols and code objects, and an incremental imager for dynamically forming the image of the program from the code objects. The present invention further provides an extremely efficient methodology for dynamically adding program elements to a program under execution.

6 Claims, 14 Drawing Sheets



DOCUMENT-IDENTIFIER: US 5848274 A
TITLE: Incremental byte code compilation system

---------- KWIC ---------

BSPR:
The stored components are accessed in sequence, and using a compiler, dependencies associated with a component are calculated to develop client and
reference **lists**. Finally, the components are compiled to build the computer
program using the properties of the components along with the compiler dependencies. The '901 reference thus reads on a system in which the program
is not defined in files, but in terms of components, addressable by means of
Ids. In this manner, the '901 reference teaches a system which does not have
the previously discussed translation problem, but replaces it with a number of
independently addressable (and reasonably complex) components, the building and
utilization of which are a major source of computational overhead. These
contribute to the lack of computational efficiency of this system.

BSPR:
Heretofore, the solution to the previously defined problem has been to use a
byte code representation of the program, which is both compact and target-independent, and then have an interpreter at the client "execute" the
byte code. This process is very inefficient and slows distributed applications
considerably. This inefficiency has, to some extent, been remedied by the use
of byte code compilation, which takes the byte code at the client and compiles
it, one function at a time, to the target architecture code. The use of byte
code compilation adds time required prior to starting the application, but
makes the application run considerably faster once compiled. The overall
result is a smaller total execution time for all but the most trivial applications.

DRPR:
FIG. 14 is a diagrammatic representation of the sequential formation of code
objects, relocation **arrays** and run-time code objects responsive to an IL
symbol.

DEPR:

Referring now to FIG. 5, the additional properties of a Declaration-ProgramUnit include a **list** of the ProgramObjects contained in the Declaration-ProgramUnit.

DEPR:
With respect to FIGS. 10-12, the Dependents field of the program objects (e.g., 1110) and the depends on field of the program units (e.g., 1160) are implemented, in a preferred embodiment, as a binary tree of pointers arranged for efficient sorting. For the sake of clarity in these figures, the actual **list** of pointers, and the several pointers referred to therein, have been omitted from the figure elements representing program units. The concept of a field comprising a binary tree of pointers is well known to those of ordinary skill in the art.

DEPR:
The Imager/Debugger not only produces information for the hardware, it also produces information for the Runner. In that process, it need not worry about relocation forms and other complexities, since all the Imager/Debugger needs is to make sure to use addresses from category c whenever it communicates with the Runner. In some circumstances where space is of concern, and it is necessary to **avoid duplication** of information, the Runner will in fact accept addresses from category b, and itself translate them to category c.

DEPR:
Code object pointer 2004 points to a code object 2010 corresponding to IL symbol 2000. Each code object 2010 contains a size field 2011, a relocations field 2012 and a code section, 2013. Code sections 2013 contain the previously discussed fully translated machine-language implementations of function definitions and the initial values of variables. Relocations field 2012 defines a relocations pointer 2014, which points to a relocations **array** 2020. Relocations **array** 2020 comprises a vector of relocation offsets and IL symbol references.

DEPR:
Translation of a class file occurs in two parts. The first part ("declaration") processes declarative information and creates symbol table.
The second part ("compilation")processes executable code (bytecode) and creates

IL. The class-file translator maintains a **list** of class files requiring
compilation (the compilation **list**). Translation begins with the declaration of
a particular class file.

DEPV:
A **list** of the Macros contained in the SourceFileProgramUnit; and

DEPV:
A **list** of the DeclarationProgramUnits which it contains.

DEPV:
3. Construct a table T containing all of the ProgramObjects in all the ProgramUnits in SF2, **indexed** by Signature.

DEPV:
4. Traverse the Program Objects in SF1. (This requires traversing the **list** of
DeclarationProgramUnits.) For each Program Object O, look up O's Signature in
T. If an entry O1 exists, copy each element of O's Dependents set to O1's
Dependents set and delete O1 from T. If an entry does not exist, mark each
element of O's Dependents set as needing rebuilding by setting the RequiresCompilation flag to True.

DEPV:
2. Compile the text of the Declaration Program Unit to produce a **list** of
Program Objects.

DEPV:
3. Construct a table T containing all of the new Program Objects, **indexed** by
Signature.

DEPV:
4. Traverse the **list** of the old Program Objects. For each Program Object O,
look up O's Signature in T. If an entry O1 exists, copy O's Dependents set to
O1 and delete O1 from T. If an entry does not exist, mark each element of O's
Dependents set as needing recompilation.

DEPV:
5. Replace the old **list** of Program Objects with the new one.

DETL:
_____ For the class: Translation of class
file: Declaration of the class Do while the compilation **list** is non-empty:
Compile the first class on the **list.** Remove the first class from the **list.**

End do End _____

DETL:
_____ Declaration of the class:
Declare the
superclass of the class. Do for each data member and method member of
the
class: Declare all classes used in the member's type. Create symbol
table
objects to represent the member. End do Create data structures (e.g.,
virtual
function tables) that support execution of the class's code. Put the
class at
the end of the compilation **list.** End Compile the class file: For
each method
defined in the class file: For each bytecode operator in the method:
If the
operator makes reference to a class Declare the class. End if;
Generate an
IL operator (or operators) with a meaning equivalent to the bytecode
operator.
If the bytecode operator expects one or more operands to come from
the VM
stack, pop the equivalent operands from the expression stack. If the
bytecode
operator leaves a result on the VM stack, push the corresponding IL
expression
on the expression stack. If the bytecode performs an imperative
operation,
produce the corresponding IL instruction. End for; Supply the
synthesized IL
to the code generator which creates a code object representing the
definition
of the method. End for; End. _____